

SEA: String Executability Analysis by Abstract Interpretation

Vincenzo Arceri¹ Mila Dalla Preda² Roberto Giacobazzi³
Isabella Mastroeni⁴

Department of Computer Science, University of Verona, Italy

Abstract

Dynamic languages often employ reflection primitives to turn dynamically generated text into executable code at run-time. These features make standard static analysis extremely hard if not impossible because its essential data structures, i.e., the control-flow graph and the system of recursive equations associated with the program to analyse, are themselves dynamically mutating objects. We introduce SEA, an abstract interpreter for automatic sound string executability analysis of dynamic languages employing bounded (i.e, finitely nested) reflection and dynamic code generation. Strings are statically approximated in an abstract domain of finite state automata with basic operations implemented as symbolic transducers. SEA combines standard program analysis together with string executability analysis. The analysis of a call to reflection determines a call to the same abstract interpreter over a code which is synthesised directly from the result of the static string executability analysis at that program point. The use of regular languages for approximating dynamically generated code structures allows SEA to soundly approximate safety properties of self modifying programs yet maintaining efficiency. Soundness here means that the semantics of the code synthesised by the analyser to resolve reflection over-approximates the semantics of the code dynamically built at run-time by the program at that point.

Keywords: Automata, Symbolic Transducers, Abstract Interpretation, Program Analysis, Dynamic Languages.

1 Introduction

Motivation.

The possibility of dynamically build code instructions as the result of text manipulation is a key aspect in dynamic programming languages. With reflection, programs can turn text, which can be built at run-time, into executable code [32]. These features are often used in code protection and tamper resistant applications, employing camouflage for escaping attack or detection [29], in malware, in mobile

¹ Email: vincenzo.arceri@univr.it

² Email: mila.dallapreda@univr.it

³ Email: roberto.giacobazzi@univr.it

⁴ Email: isabella.mastroeni@univr.it

```

1  //Retrieve the ID for a camera (e.g., the front-facing camera)
2  int cameraId = ...;

3  //Create an obfuscated string containing the method to call ("open")
4  String obfuscated = "koOpqUTbcVRhwomXlASpvutejuWHJnQxxaoinoermf";

5  String deobfuscated =
6  obfuscated.replaceAll( "[RhwmXlASvutjWHJQxa]", "").substring(10, 14);
7  // Now deobfuscated contains "open"

8  Class<?> klass = Class.forName( "android.hardware.Camera");

9  //Retrieve and invoke the method
10 Method method = klass.getMethod(deobfuscated, Integer.class);

11 Camera camera = (Camera) method.invoke(cameraId);

12 //use the camera to take a picture
13 ...

```

Fig. 1. A template ransomware obfuscated attack

code, in web servers, in code compression, and in code optimisation, e.g., in Just-in-Time (JIT) compilers employing optimised run-time code generation.

While the use of dynamic code generation may simplify considerably the *art and performance of programming*, this practice is also highly dangerous, making the code prone to unexpected behaviour and malicious exploits of its dynamic vulnerabilities, such as code/object-injection attacks for privilege escalation, data-base corruption, and malware propagation. It is clear that more advanced and secure functionalities based on reflection could be permitted if we better master how to safely generate, analyse, debug, and deploy programs that dynamically generate and manipulate code.

There are lots of good reasons to analyse when a program builds strings that can be later executed as code. Consider the code in Fig. 1. This is a template of a ransomware that calls a method (“open”) by manipulating an obfuscated string which is built in the code. Analysing the flow of the strings corresponds here to approximate the set of strings that may be turned into code at run-time. This possibility would provide important benefits in the analysis of dynamic languages, without ignoring reflection, in automated deobfuscation of dynamic obfuscators, and in the analysis of code injection and XSS attacks.

The problem.

A major problem in dynamic code generation is that static analysis becomes extremely hard if not even impossible. This because program’s essential data structures, such as the control-flow graph and the system of recursive equations associated with the program to analyse, are themselves dynamically mutating objects. In a sentence: *“You can’t check code you dont see”* [5].

The standard way for treating dynamic code generation in programming is to prevent or even banish it, therefore restricting the expressivity of our development tools. Other approaches instead tries to combine static and dynamic analysis to predict the code structures dynamically generated [41,7]. Because of this difficulty, most analyses of dynamic languages do not consider reflection [3], thus being inherently unsound for these languages, or implement ad-hoc or pattern driven trans-

formations in order to remove reflection [26]. The design and implementation of a sound static analyser for self mutating programs is still nowadays an open challenge for static program analysis.

Contribution.

In this paper we solve this problem by treating the code as any other dynamic structure that can be statically analysed by abstract interpretation [13]. We introduce SEA, a proof of concept for a fully automatic sound-by-construction abstract interpreter for string executability analysis of dynamic languages employing finitely nested (bounded) reflection and dynamic code generation. SEA carries a generic standard numerical analysis, in our case a well-known interval analysis, together with a new string executability analysis. Strings are approximated in an abstract domain of finite state automata (FA) with basic operations implemented as symbolic transducers and widening for enforcing termination.

The idea in SEA is to re-factor reflection into a program whose semantics is a sound over-approximation of the semantics of the dynamically generated code. This allows us to continue with the standard analysis when the reflection is called on an argument that evaluates to code. In order to recognise whether approximated strings correspond to executable instructions, we approximate a parser as a symbolic transducer and, in case of success, we synthesise a program from the FA approximation of the computed string. The static analysis of reflection determines a call to the same abstract interpreter over the code synthesised from the result of the static string executability analysis at that program point. The choice of regular languages for approximating code structures provides efficient implementations of both the analysis and the code generation at analysis time. The synthesised program reflects most of the structures of the code dynamically generated, yet losing some aspects such as the number of iterations of loops.

Soundness here means that, if the approximated code extracted by the abstract interpreter is accepted by the parser, then the program may dynamically generate executable code at run-time. Moreover, because of the approximation of dynamically generated code structures in a regular language of instructions, any sound and terminating abstract interpretation for safety (i.e., prefix closed) properties of the synthesised code, over-approximates the concrete semantics of the dynamically generated code. This means that a sound over-approximation of the concrete semantics of programs dynamically generating and executing code by reflection is possible for safety properties by combining string analysis and code synthesis in abstract interpretation. Even if nested reflection is not a common practice, the case of potentially unbound nested reflections, which may lead to non termination the analysis, can be handled as in [26] by fixing a maximal degree of nesting allowed. In this case, for programs exceeding the maximal degree of nested reflections, we may lose soundness. We briefly discuss how in SEA we may achieve an always sound analysis also for unbound reflection based on a widening with threshold over the recursive applications of the abstract interpreter.

2 Related Works

The analysis of strings is nowadays a relatively common practice in program analysis due to the widespread use of dynamic scripting languages. Examples of analyses for string manipulation are in [20,10,40,34,30,28]. The use of symbolic (grammar-based) objects in abstract domains is also not new (see [15,23,36]) and some works explicitly use transducers for string analysis in script sanitisation, see for instance [25] and [40], all recognising that specifying the analysis in terms of abstract interpretation makes it suitable for being combined with other analyses, with a better potential in terms of tuning in accuracy and costs. None of these works use string analysis for analysing executability of dynamically generated code. In [26], the authors introduce an automatic code rewriting techniques removing `eval` constructs in JavaScript applications. This work has been inspired by the work of Richards et al. [32] showing that `eval` is widely used, nevertheless in many cases its use can be simply replaced by JavaScript code without `eval`. In [26] the authors integrate a refactoring of the calls to `eval` into the TAJIS data-flow analyzer. TAJIS performs inter-procedural data-flow analysis on an abstract domain of objects capturing whether expressions evaluate to constant values. In this case `eval` calls can be replaced with an alternative code that does not use `eval`. It is clear that code refactoring is possible only when the abstract analysis recognises that the arguments of the `eval` call are constants. Moreover, they handle the presence of nested `eval` by fixing a maximal degree of nesting, but in practice they set this degree to 1, since, as they claim, it is not often encountered in practice. The solution we propose allows us to go beyond constant values and refactor code also when then arguments of `eval` are elements of a regular language of strings. While this can be safely used for analysing safety properties of dynamically generated code, the use of our method for code refactoring has to take into account non-terminating code introduced by widening and regular language approximation. A more detailed comparison with TAJIS is discussed in Sect. 7.

Static analysis for a static subset of PHP (i.e., ignoring `eval`-like primitives) has been developed in [6]. Static *taint analysis* keeping track of values derived from user inputs has been developed for self-modifying code by partial derivation of the Control-Flow-Graph [38]. The approach is limited to taint analysis, e.g., for limiting code-injection attacks. Staged information flow for JavaScript in [11] with *holes* provides a conditional (a la abduction analysis in [22]) static analysis of dynamically evaluated code. Symbolic execution-based static analyses have been developed for scripting languages, e.g., PHP, including primitives for code reflection, still at the price of introducing false negatives [39].

We are not aware of effective general purpose sound static analyses handling self-modifying code for high-level scripting languages. On the contrary, a huge effort was devoted to bring static type inference to object-oriented dynamic languages (e.g., see [3] for an account in Ruby) but with a different perspective: *Bring into dynamic languages the benefits of static ones – well-typed programs don’t go wrong*. Our approach is different: *Bring into static analysis the possibility of handling dynamically mutating code*. A similar approach is in [4] and [16] for binary executables.

The idea is that of extracting a code representation which is descriptive enough to include most code mutations by a dynamic analysis, and then reform analysis on a linearization of this code. On the semantics side, since the pioneering work on certifying self-modifying code in [9], the approach to self-modifying code consists in treating machine instructions as regular mutable data structures, and to incorporate a logic dealing with code mutation within a la Hoare logics for program verification. TamiFlex [7] also synthesises a program at every `eval` call by considering the code that has been executed during some (dynamically) observed execution traces. The static analysis can then proceed with the so obtained code without `eval`. It is sound only with respect to the considered execution traces, producing a warning otherwise.

3 Preliminaries

Mathematical Notation.

S^* is the set of all finite sequences of elements in S . We often use bold letters to denote them. If $\mathbf{s} = s_1 \dots s_n \in S^*$, $s_i \in S$ is the i -th element and $|\mathbf{s}| \in \mathbb{N}$ its length. If $\mathbf{s}_1, \mathbf{s}_2 \in \Sigma^*$, $\mathbf{s}_1 \cdot \mathbf{s}_2 \in \Sigma^*$ is their concatenation.

A set L with ordering relation \leq is a poset and it is denoted as $\langle L, \leq \rangle$. Lattices L with ordering \leq , least upper bound (lub) \vee , greatest lower bound (glb) \wedge , greatest element (top) \top , and least element (bottom) \perp are denoted $\langle L, \leq, \vee, \wedge, \top, \perp \rangle$. Given $f : S \rightarrow T$ and $g : T \rightarrow Q$ we denote with $g \circ f : S \rightarrow Q$ their composition, *i.e.*, $g \circ f = \lambda x. g(f(x))$. For $f, g : L \rightarrow D$ on complete lattices $f \sqcup g$ denotes the point-wise least upper bound, *i.e.*, $f \sqcup g = \lambda x. f(x) \vee g(x)$. f is *additive* (*co-additive*) if for any $Y \subseteq L$, $f(\vee_L Y) = \vee_D f(Y)$ ($f(\wedge_L Y) = \wedge_D f(Y)$). The additive lift of a function $f : L \rightarrow D$ is the function $\lambda X \subseteq L. \{f(x) \mid x \in X\} \in \wp(D)$. We will often identify a function and its additive lift. Continuity holds when f preserves *lubs*'s of chains. For a continuous function f : $\text{lfp}(f) = \bigwedge \{x \mid x = f(x)\} = \bigvee_{n \in \mathbb{N}} f^n(\perp)$ where $f^0(\perp) = \perp$ and $f^{n+1}(\perp) = f(f^n(\perp))$.

Abstract Interpretation.

Abstract interpretation establishes a correspondence between a concrete semantics and an approximated one called abstract semantics [13,14]. In a Galois Connection (GC) framework, if C and A are complete lattices, a pair of monotone functions $\alpha : C \rightarrow A$ and $\gamma : A \rightarrow C$ forms a GC between C and A if for every $x \in C$ and $y \in A$ we have $\alpha(x) \leq_A y \Leftrightarrow x \leq_C \gamma(y)$. α (resp. γ) is the *abstraction* (resp. *concretisation*) and it is additive (resp. co-additive). Weaker forms of correspondence are possible, e.g., when A is not a complete lattice or when only γ exists. In all cases, relative precision in A is given by comparing the meaning of abstract objects in C , *i.e.*, $x_1 \leq_A x_2$ if $\gamma(x_1) \leq_C \gamma(x_2)$. If $f : C \rightarrow C$ is a continuous function and A is an abstraction of C by means of the GC $\langle \alpha, \gamma \rangle$, then f always has a *best correct approximation* in A , $f^A : A \rightarrow A$, defined as $f^A \triangleq \alpha \circ f \circ \gamma$. Any approximation $f^\# : A \rightarrow A$ of f in A is *sound* if $f^A \sqsubseteq f^\#$. In this case we have the fix-point soundness $\alpha(\text{lfp} f) \leq \text{lfp}(f^A) \leq \text{lfp}(f^\#)$ (cf. [13]). A satisfies the ascending chain condition

(ACC) if all ascending chains are finite. When A is not ACC or when it lacks the limits of chains, convergence to the limit of the fix-point iterations can be ensured through widening operators. A *widening operator* $\nabla : A \times A \rightarrow A$ approximates the lub, i.e., $\forall x, y \in A. x, y \leq_A (x \nabla y)$ and it is such that for any increasing chain $x_1 \leq x_2 \leq \dots \leq x_n \leq \dots$ the increasing chain $w^0 = \perp$ and $w^{i+1} = w^i \nabla x_i$ is finite.

Finite State Automata (FA).

A FA A is a tuple $(Q, \delta, q_0, F, \Sigma)$, where Q is the set of states, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states and Σ is the finite alphabet of symbols. An element $(q, \sigma, q') \in \delta$ is called transition and is denoted $q' \in \delta(q, \sigma)$. Let $\omega \in \Sigma^*$, $\hat{\delta} : Q \times \Sigma^* \rightarrow \wp(Q)$ is the transitive closure of δ : $\hat{\delta}(q, \epsilon) = \{q\}$ and $\hat{\delta}(q, \omega\sigma) = \bigcup_{q' \in \delta(q, \omega)} \delta(q', \sigma)$. $\omega \in \Sigma^*$ is accepted by A if $\hat{\delta}(q_0, \omega) \cap F \neq \emptyset$. The set of all these strings defines the language $\mathcal{L}(A)$ accepted by A . Given an FA A and a partition π over its states, we denote as $A/\pi = (Q', \delta', q'_0, F', \Sigma)$ the *quotient automaton* [19].

Symbolic Finite Transducers (SFT).

We follow [35] in the definition of SFTs and of their background structure. Consider a background universe \mathcal{U}_τ of elements of type τ , we denote with \mathbb{B} to denote the elements of boolean type. Terms and formulas are defined by induction over the background language and are well-typed. Terms of type \mathbb{B} are treated as formulas. $t : \tau$ denotes a term t of type τ , and $FV(t)$ denotes the set of its free variables. A term $t : \tau$ is *closed* when $FV(t) = \emptyset$. Closed terms have semantics $\llbracket t \rrbracket$. As usual $t[x/v]$ denotes the substitution of a variable $x : \tau$ with a term $v : \tau$. A λ -term f is an expression of the form $\lambda x. t$ where $x : \tau'$ is a variable and $t : \tau''$ is a term such that $FV(t) \subseteq \{x\}$. The λ -term f has type $\tau' \rightarrow \tau''$ and its semantics is a function $\llbracket f \rrbracket : \mathcal{U}_{\tau'} \rightarrow \mathcal{U}_{\tau''}$ that maps $a \in \mathcal{U}_{\tau'}$ to $\llbracket t[x/a] \rrbracket \in \mathcal{U}_{\tau''}$. Let f and g range over λ -terms. A λ -term of type $\tau \rightarrow \mathbb{B}$ is called a τ -predicate. Given a τ -predicate φ , we write $a \in \llbracket \varphi \rrbracket$ for $\llbracket \varphi \rrbracket(a) = \text{true}$. Moreover, $\llbracket \varphi \rrbracket$ can be seen as the subset of \mathcal{U}_τ that satisfies φ . φ is *unsatisfiable* when $\llbracket \varphi \rrbracket = \emptyset$ and *satisfiable* otherwise. A label theory [35] for $\tau' \rightarrow \tau''$ is associated with an effectively enumerable set of λ -terms of type $\tau' \rightarrow \tau''$ and an effectively enumerable set of τ' -predicates that is effectively closed under Boolean operations and relative difference, i.e., $\llbracket \varphi \wedge \psi \rrbracket = \llbracket \varphi \rrbracket \cap \llbracket \psi \rrbracket$, and $\llbracket \neg \varphi \rrbracket = \mathcal{U}_{\tau'} \setminus \llbracket \varphi \rrbracket$. Let τ^* be the type of sequences of elements of type τ . A Symbolic Finite Transducer [35] (SFT) T over $\tau' \rightarrow \tau''$ is a tuple $T = \langle Q, q^0, F, R \rangle$, where Q is a finite set of states, $q^0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states and R is a set of rules $(p, \varphi, \mathbf{f}, q)$ where $p, q \in Q$, φ is a τ' -predicate and \mathbf{f} is a sequence of λ -terms over a given label theory for $\tau' \rightarrow \tau''$. A rule $(p, \varphi, \mathbf{f}, q)$ of an SFT T is denoted as $p \xrightarrow{\varphi/\mathbf{f}} q$. The sequence of λ -terms $\mathbf{f} : (\tau' \rightarrow \tau'')^*$ can be treated as a function $\lambda x. [\mathbf{f}_0(x), \dots, \mathbf{f}_k(x)]$ where $k = |\mathbf{f}| - 1$. Concrete transitions are represented as rules. Let $p, q \in Q$, $a \in \mathcal{U}_{\tau'}$ and $\mathbf{b} \in \mathcal{U}_{\tau''}^*$, then: $\xrightarrow{a/\mathbf{b}}_T q \Leftrightarrow p \xrightarrow{\varphi/\mathbf{f}}_T q \in R : a \in \llbracket \varphi \rrbracket \wedge \mathbf{b} = \llbracket \mathbf{f} \rrbracket(a)$ Given two sequences $\mathbf{a} \in \mathcal{U}_{\tau'}^*$ and $\mathbf{b} \in \mathcal{U}_{\tau''}^*$, we write $q \xrightarrow{\mathbf{a}/\mathbf{b}} p$ when there exists a path

of transitions from q to p in T with input sequence $\mathbf{a} = \mathbf{a}_0\mathbf{a}_1\cdots\mathbf{a}_n$ and output sequence $\mathbf{b} = \mathbf{b}^0\mathbf{b}^1\cdots\mathbf{b}^n$, $n = |\mathbf{a}| - 1$ and \mathbf{b}^i denoting a subsequence of \mathbf{b} , such that: $p = p_0 \xrightarrow{\mathbf{a}_0/\mathbf{b}^0} p_1 \xrightarrow{\mathbf{a}_1/\mathbf{b}^1} p_2 \cdots p_n \xrightarrow{\mathbf{a}_n/\mathbf{b}^n} p_{n+1} = q$. SFT can have ε -transitions and they can be eliminated following a standard procedure. We assume $p \xrightarrow{\varepsilon/\varepsilon} p$ for all $p \in Q$. The transduction of an SFT T [35] over $\tau' \rightarrow \tau''$ is a function $\mathfrak{T}_T : \mathcal{U}_{\tau'}^* \rightarrow \wp(\mathcal{U}_{\tau''}^*)$ where: $\mathfrak{T}_T(\mathbf{a}) \triangleq \{\mathbf{b} \in \mathcal{U}_{\tau''}^* \mid \exists q \in F : q^0 \twoheadrightarrow q\}$

SFT as FA Transformers.

In the following, we will consider SFTs producing only one symbol in output for each symbol read in input. Namely, we consider SFTs with rules (q, φ, f, q) where f is a single λ -term of type $\tau' \rightarrow \tau''$. Moreover, we consider SFTs and FA over finite alphabets, where the symbolic representation of SFT is useful for having more compact language transformers.

In this section we show how, under these assumptions, SFTs can be seen as FA transformers. In particular, given an FA A such that $\mathcal{L}(A) \in \wp(\mathcal{U}_{\tau'}^*)$ and an SFT T over $\tau' \rightarrow \tau''$, we want to build the FA recognizing the language of strings in $\mathcal{U}_{\tau''}^*$ obtained by modifying the strings in $\mathcal{L}(A)$, according to the SFT T . To this end, we define the input language $\mathcal{L}_{\mathcal{I}}(T)$ of an SFT T as the set of strings producing an output when processed by T , and the output language $\mathcal{L}_{\mathcal{O}}(T)$ as the set of strings generated by T . Formally: $\mathcal{L}_{\mathcal{I}}(T) \triangleq \{\mathbf{a} \in \mathcal{U}_{\tau'}^* \mid \mathfrak{T}_T(\mathbf{a}) \neq \emptyset\}$ and $\mathcal{L}_{\mathcal{O}}(T) \triangleq \{\mathbf{b} \in \mathcal{U}_{\tau''}^* \mid \mathbf{b} \in \mathfrak{T}_T(\mathbf{a}), \mathbf{a} \in \mathcal{L}_{\mathcal{I}}(T)\}$.

Consider $T = \langle Q, q^0, F, R \rangle$ over $\tau' \rightarrow \tau''$, with $\mathcal{U}_{\tau'}$ and $\mathcal{U}_{\tau''}$ finite alphabets, and rules $(q, \varphi, f, p) \in R$ such that f are λ -terms of type $\tau' \rightarrow \tau''$. According to [17] it is possible to build an FA $\text{FA}_{\mathcal{O}}(T)$ recognising the output language of T , i.e., $\mathcal{L}(\text{FA}_{\mathcal{O}}(T)) = \mathcal{L}_{\mathcal{O}}(T)$. In particular, $\text{FA}_{\mathcal{O}}(T) \triangleq (Q, \delta, q_0, F, \mathcal{U}_{\tau''})$ where $\delta = \{(q, b, p) \mid (q, \varphi, f, p) \in R, b \in \llbracket f(\varphi) \rrbracket\}$. Observe that $\llbracket f(\varphi) \rrbracket$ is finite since φ is a predicate over a finite alphabet. We can associate an SFT $\mathcal{T}(A)$ to an FA A , where the input and output languages of $\mathcal{T}(A)$ are the ones recognized by the FA A . Formally, given an FA $A = (Q, \delta, q_0, F, \mathcal{U}_{\tau})$, we define the output SFT over $\tau \rightarrow \tau$ as $\mathcal{T}(A) \triangleq \langle Q, q_0, F, R^{\text{id}} \rangle$ where $R^{\text{id}} \triangleq \{(p, \sigma, \text{id}, q) \mid (p, \sigma, q) \in \delta\}$ ⁵ and the transduction is:

$$\mathfrak{T}_{\mathcal{T}(A)}(\mathbf{a}) = \begin{cases} \mathbf{a} & \text{if } \mathbf{a} \in \mathcal{L}(A) \\ \emptyset & \text{otherwise} \end{cases}$$

These definitions allow us to associate FAs with SFTs and vice-versa and. According to [35], we define the composition of two transductions \mathfrak{T}_1 and \mathfrak{T}_2 as:

$$\mathfrak{T}_1 \diamond \mathfrak{T}_2 \triangleq \lambda \mathbf{b}. \bigcup_{\mathbf{a} \in \mathfrak{T}_1(\mathbf{b})} \mathfrak{T}_2(\mathbf{a})$$

Observe that the composition \diamond applies first \mathfrak{T}_1 and then \mathfrak{T}_2 . It has been proved that if T_1 and T_2 are SFTs over composable label theories, then there exists an SFT $T_1 \diamond T_2$ that is obtained effectively from T_1 and T_2 such that $\mathfrak{T}_{T_1 \diamond T_2} = \mathfrak{T}_1 \diamond \mathfrak{T}_2$ (see

⁵ We denote by σ the predicate requiring the symbol to be equal to σ .


```

Exp  $\ni$  e ::= a | b | s
AExp  $\ni$  a ::= x | n | rand() | len(s) | num(s) |
           a + a | a - a | a * a      (where  $n \in \mathbb{Z}$ )
BExp  $\ni$  b ::= x | tt | ff | e = e | e > e | e < e | b  $\wedge$  b |  $\neg$ b
SExp  $\ni$  s ::= x | ' ' | 'σ' | s . σ | substr(s, a, a)
           (where  $\sigma \in \Sigma$ )
Comm  $\ni$  c ::= skip; | x := e; | cc | if b {c}; |
           while b {c}; | reflect(s); | x := reflect(s);
Dlmp  $\ni$  P ::= c$
Id  $\ni$  x   Identifiers (strings not containing punctuation symbols)

```

Fig. 2. Syntax of Dlmp

[35,24] for details). At this point, given an FA A with $\mathcal{L}(A) \in \wp(\mathcal{U}_{\tau'}^*)$ and an SFT T over $\tau' \rightarrow \tau''$, we can model the application of T to $\mathcal{L}(A)$ as the composition $\mathcal{T}(A) \diamond T$ where the language recognized by the FA A becomes the input language of the SFT T . $\mathfrak{T}_{\mathcal{T}(A) \diamond T} = \mathfrak{T}_T(\mathbf{b})$ if $\mathbf{b} \in \mathcal{L}(A)$, it is \emptyset otherwise. Observe that, the FA recognizing the output language of $\mathcal{T}(A) \diamond T$ is the FA obtained by transforming A with T . Indeed, $\mathcal{L}(\text{FA}_{\mathcal{O}}(\mathcal{T}(A) \diamond T)) = \{\mathbf{b} \in \Gamma^* \mid \mathbf{b} \in \mathfrak{T}_T(\mathbf{a}), \mathbf{a} \in \mathcal{L}(A)\}$. Thus, we can say that an SFT T transforms an FA A into the FA $\text{FA}_{\mathcal{O}}(\mathcal{T}(A) \diamond T)$.

4 A Core Dynamic Programming Language

4.1 The dynamic language

We introduce a core imperative deterministic dynamic language **Dlmp**, in the style of **IMP** for its imperative fragment and of dynamic languages, such as **PHP** or **JavaScript**, as far as string manipulation is concerned, with basic types integers in \mathbb{Z} , booleans, and strings of symbols over a finite alphabet Σ . Programs P are labeled commands in **Dlmp** built as in Figure 2, on a set of variables Var and line of code labels PLines_P with typical elements $l \in \text{PLines}_P$. We assume that all terminal and non terminal symbols of **Dlmp** are in $\Sigma_{\text{Dlmp}} \subseteq \Sigma^*$. Thus, the language recognized by the context free grammar (CFG) of **Dlmp** is an element of $\wp((\Sigma_{\text{Dlmp}})^*)$, i.e., $\text{Dlmp} \subseteq (\Sigma_{\text{Dlmp}})^*$. Given $P \in \text{Dlmp}$ we associate with each statement a program line $l \in \text{PLines}_P$. In order to simplify the presentation of the semantics, we suppose that any program is ended by a termination symbol $\$$, labeled with the last program line denoted l_e . When a statement c belongs to a program P we write $c \in P$, then we define the auxiliary functions $\text{Stm}_P : \text{PLines}_P \rightarrow \text{Dlmp}$ be such that $\text{Stm}_P(l) = c$ if c is the statement in P at program line l (in the following denoted ${}^l c$) and $\text{Pl}_P = \text{Stm}_P^{-1} : \text{Dlmp} \rightarrow \text{PLines}_P$ with the simple extension to blocks of P instructions $\text{Pl}_P(c_1 c_2) = \text{Pl}_P(c_1)$. In general, we denote by Pl_P the set of all the program lines in P ⁶.

Let $M \triangleq \text{Var} \longrightarrow \mathbb{Z} \cup \{\text{tt}, \text{ff}\} \cup \Sigma^*$ be the set of memory maps, ranged over by m , that assign values (integers, booleans or strings) to variables. $\langle \mathbf{s} \rangle : M \longrightarrow \Sigma^*$

⁶ Note that, by definition a statement, or a block, c ends always with $;$.

denotes the semantics of string expressions. For strings $s_1, s_2 \in \Sigma^*$, symbol $\delta \in \Sigma$ and values $n_1, n_2 \in \mathbb{Z}$ we have that $\llbracket s_1 \cdot \delta \rrbracket m$ returns the concatenation of the string $\llbracket s_1 \rrbracket m$ with the symbol $\delta \in \Sigma$, i.e., $\llbracket s_1 \rrbracket m \cdot \delta$. We abuse notation and use $s_1 \cdot s_2$ for string concatenation. The semantics $\llbracket \text{substr}(s, n_1, n_2) \rrbracket m$ returns the sub-string of the string $\llbracket s \rrbracket m$ given by the n_2 consecutive symbols starting from the n_1 -th one (we suppose $n_1 \geq 0$)⁷. We denote with $\llbracket a \rrbracket : M \rightarrow \mathbb{Z}$ the semantics of arithmetic expressions where $\llbracket \text{len}(s) \rrbracket m$ returns the length of the string $\llbracket s \rrbracket m$, and $\llbracket \text{num}(s) \rrbracket m$ returns the integer denoted by the string $\llbracket s \rrbracket m$ (suppose it returns the empty set if $\llbracket s \rrbracket m$ is not a number). The semantics of the other arithmetic expressions is defined as usual. Analogously, $\llbracket b \rrbracket : M \rightarrow \{\text{tt}, \text{ff}\}$ denotes the semantics of Boolean expressions where, given $s_1, s_2 \in \Sigma^*$, $s_2 < s_1$ is true iff $s_2 \preceq s_1$ (prefix order). The semantics of the other Boolean expressions is defined as usual.

The update of memory m , for a variable x with value v , is denoted $m[x/v]$. The semantics of **reflect**(s) evaluates the string s : if it is a program in **DImp** it executes it, otherwise the execution proceeds with the next command. Observe that $s \in \Sigma^*$ while $\text{DImp} \in \wp((\Sigma_{\text{DImp}})^*)$, for this reason we define $\overline{\text{DImp}} \triangleq \{a \in \Sigma^* \mid a = a_1 \cdot a_2 \cdot \dots \cdot a_n, a_1 a_2 \dots a_n \in \text{DImp}\}$ as the set of sequences in Σ^* that can be obtained by concatenating the sequences a^i that act like symbols in a program in **DImp**. We denote with \bar{c} the sequence of Σ^* that corresponds to the sequence $c \in (\Sigma_{\text{DImp}})^*$. At this point, before computing the semantics of \bar{c} , we need to recognize which statements it denotes, building the corresponding string $c \in \text{DImp}$, and then to label this statements by using the function $\text{lab}(\cdot)$, assigning an integer label to each statement in $c\$$ from 1 to the final program point l_e . In the following, we say that s evaluates to c when it assumes a value $\bar{c} \in \overline{\text{DImp}}$ corresponding to the concatenation of the sequences that are symbols of c . The semantics of $x := \text{reflect}(s)$ evaluates expression s and if it is c in **DImp** it proceeds by assigning $''$ to x and executes c , otherwise it behaves as a standard assignment. Formally, let $\text{Int} : \text{DImp} \times M \rightarrow M$ denote the semantics of programs, and $\llbracket \cdot \rrbracket m$ the evaluation of an expression in the memory m , then:

$$\text{Int}(l \cdot \text{reflect}(s); l' \cdot Q, m) = \begin{cases} \text{Int}(l' \cdot Q, m') & \text{if } \llbracket s \rrbracket m \cap \overline{\text{DImp}} = \bar{c} \wedge \\ & m' = \text{Int}(\text{lab}(c), m) \\ \text{Int}(l' \cdot Q, m) & \text{otherwise} \end{cases}$$

We can observe that the way in which we treat the commands $l \cdot \text{reflect}(s)$ and $l \cdot x := \text{reflect}(s)$ mimics the classical semantic model and implementation of reflection and reification as introduced in Smith [33], see [37, 18] for details. In particular, when string s evaluates to a program c , the program control starts the execution of c before returning to the original code. The problem is that c may contain other **reflect** statements leading to the execution of new portions of code. Hence, each nested **reflect** is an invocation of the interpreter which can be seen as a new layer in the *tower* of interpretations: When a layer terminates the execution the control returns to the previous layer with the actual state. Hence, the

⁷ The choice of considering only concatenation and substring derives from the fact that most of the operations on strings can be obtained as by using these two operations.

state $\text{Int}({}^l\mathbf{reflect}(\mathbf{s}); {}^{l'}\mathbf{Q}, m)$, when string \mathbf{s} evaluates to a program \mathbf{c} , starts a new computation of $\text{lab}(\mathbf{c})$ from m . Once the execution of the tower derived from \mathbf{c} terminates, the execution comes back to the continuation \mathbf{Q} , in the memory resulting from the execution of \mathbf{c} . It is known that in general the construction of the tower of interpreters may be infinite leading to a divergent semantics.

Example 4.1. Consider the following program fragment \mathbf{P} :

$${}^1.x := \mathbf{reflect}(x); \$'; {}^2.\mathbf{reflect}(x); {}^3.\$$$

Suppose the initial memory is m_\perp (associating the undefined value to each variable, in this case x). After the execution of the first assignment we have the memory $m_1 = [x/\mathbf{reflect}(x)']$, on which we execute the $\mathbf{reflect}(x)$ statement. Since now, $\mathbf{reflect}(x)$ is executed starting from m_1 , hence each $\mathbf{reflect}(x)$ activates a tower layer executing the statement in x , which is again $\mathbf{reflect}(x)$ starting from the same memory. Hence the tower has infinite height.

4.2 Flow-sensitive Collecting Semantics

Collecting semantics models program execution by computing, for each program point, the set of all the values that each variables may have. In order to deal with reflection we need to define an interpreter collecting values for each program point which, at each step of computation, keeps trace not only of the collection of values after the last executed program point p , but also of the values collected in all the other program points, both already executed and not executed yet. In other words, we define a flow sensitive semantics which, at the end of the computation, observes the trace of collections of values holding *at each program point*. In order to model this semantics, we model the concrete state not simply as a memory – the current memory, but as the tuple of memories holding at each program point. It is clear that, at each step of computation, only the memory in the last executed program point will be modified. First, we define a collecting memory \mathfrak{m} , associating with each variable a set of values instead of a single value. We define the set $\mathbb{M} \triangleq \text{Var} \longrightarrow \wp(\mathbb{Z}) \cup \text{Bool} \cup \wp(\Sigma^*)$ with meta-variable \mathfrak{m} , where $\text{Bool} = \wp(\{\mathbf{ff}, \mathbf{tt}\})$. We define two particular memories, \mathfrak{m}_\emptyset associating \emptyset to any variable, and \mathfrak{m}_\top associating the set of all possible values to each variable. The update of memory \mathfrak{m} for a variable x with set of values v is denoted $\mathfrak{m}[x/v]$. Finally, lub and glb of memories are $\mathfrak{m}_1 \sqcup \mathfrak{m}_2(x) = \mathfrak{m}_1(x) \cup \mathfrak{m}_2(x)$ and $\mathfrak{m}_1 \sqcap \mathfrak{m}_2(x) = \mathfrak{m}_1(x) \cap \mathfrak{m}_2(x)$. Then, in order to make the semantics flow-sensitive, we introduce a new notion of *flow-sensitive* store (in the following called store) $\mathbb{S} \triangleq \text{PLines}_P \longrightarrow \mathbb{M}$ associating with each program line a memory. We represent a store $\mathfrak{s} \in \mathbb{S}$ at a given line $l \in \text{PLines}_P$ as a tuple $\langle x_1/v_{x_1}, \dots, x_n/v_{x_n} \rangle$, where v_{x_i} is the set of possible values of variable x_i . We use \mathfrak{s}_l to denote $\mathfrak{s}(l)$, namely the memory at line l . Given a store \mathfrak{s} , the update of memory \mathfrak{s}_l with a new collecting memory \mathfrak{m} is denoted $\mathfrak{s}[\mathfrak{s}_l \leftarrow \mathfrak{m}]$ and provides a new store \mathfrak{s}' such that $\mathfrak{s}'_l = \mathfrak{s}_l \sqcup \mathfrak{m}$ while $\forall l' \neq l$ we have $\mathfrak{s}'_{l'} = \mathfrak{s}_{l'}$.

We abuse notation by denoting with $\langle \cdot \rangle$, not only the concrete, but also the collecting semantics of expressions, all defined as additive lift of the expression

semantics. In particular, we denote by $\langle \mathbf{b} \rangle^{\mathbf{tt}}$ the maximal collecting memory making \mathbf{b} true, i.e., it is $\bigsqcup \{m \in M \mid \langle \mathbf{b} \rangle m = \mathbf{tt}\} \in \mathbb{M}$ (analogous for $\langle \mathbf{b} \rangle^{\mathbf{ff}}$). Hence, by $\mathfrak{m} \sqcap \langle \mathbf{b} \rangle^{\mathbf{tt}}$ we denote the memory $\mathfrak{m}' \triangleq \mathfrak{m}[x \in \text{vars}(\mathbf{b}) / \mathfrak{m}(x) \cap \langle \mathbf{b} \rangle^{\mathbf{tt}}(x)]$, where $\text{vars}(\mathbf{b})$ is the set of variables of \mathbf{b} . For instance if $\mathfrak{m} = [x/\{1, 2, 3\}, y/\{1, 2\}]$ and $\mathbf{b} = (x < 3)$, then $\langle \mathbf{b} \rangle^{\mathbf{tt}} = [x/\{1, 2\}, y/\top]$, hence $\mathfrak{m} \sqcap \langle \mathbf{b} \rangle^{\mathbf{tt}} = [x/\{1, 2\}, y/\{1, 2\}]$. Finally, let $V \subseteq \text{Var}$, by \mathfrak{s}_V we denote the store where for each program point l , the memory \mathfrak{s}_l is restricted only on the variables in V , by $\text{lfp}_V f(\mathfrak{s})$ we denote the computation of the fix point only on the variables V , i.e., we compute \mathfrak{s} such that $\mathfrak{s}_V = f(\mathfrak{s})_V$.

We follow [12] in the usual definition of the *concrete collecting trace semantics* of a transition system $\langle \mathbb{C}, \sim \rangle$ associated with programs in Dlmp , where $\mathbb{C} = \text{Dlmp} \times \mathbb{S}$ is the set of states in the transition system with typical elements $\mathbf{c} \in \mathbb{C}$ and $\sim \subseteq \mathbb{C} \times \wp(\mathbb{C})$ is a transition relation. The state space in the transition system is the set of all pairs $\langle \mathbf{c}, \mathfrak{s} \rangle$ with $\mathbf{c} \in \text{Dlmp}$ and $\mathfrak{s} \in \mathbb{S}$ representing the store computed by having executed the first statement in \mathbf{c} and having its continuation still to execute. The transition relation generated by a program $\mathbf{c} \in \text{Dlmp}$ is in Appendix. The axiom $\langle \text{le}, \$, \mathfrak{s} \rangle$ identifies the final blocking states \mathcal{B} . When the next command to execute is ${}^l \text{reflect}(\mathfrak{s})$, we need to verify whether the evaluation of the string \mathfrak{s} at program line l returns a set of sequences of symbols of Σ that contains sequences representing programs in Dlmp . If this is the case, we proceed by executing the programs corresponding to $\langle \mathfrak{s} \rangle \mathfrak{s}_l$ with initial memory (at the first program line of \mathbf{c}) the memory holding at program line l , while the memories for all the other program points in \mathbf{c} are initialized to \mathfrak{m}_\emptyset . When the next command to execute is an assignment of the form ${}^l x := \text{reflect}(\mathfrak{s})$ we need to verify whether string \mathfrak{s} evaluates to a simple set of strings or to strings corresponding to programs in Dlmp . If the evaluation of the strings in $\langle \mathfrak{s} \rangle \mathfrak{s}_l$ does not contain programs we proceed as for standard assignments. If $\langle \mathfrak{s} \rangle \mathfrak{s}_l$ returns programs in Dlmp then the assignment becomes an assignment of $' '$ to variable x and the execution of the programs corresponding to $\langle \mathfrak{s} \rangle \mathfrak{s}_l$. Observe that, in order to verify whether the possible values assumed by a string \mathfrak{s} at a program point l are programs in Dlmp , we check if the intersection $\langle \mathfrak{s} \rangle \mathfrak{s}_l \cap \overline{\text{Dlmp}}$ is not empty. Unfortunately, this step is in general undecidable, for this reason, in Section 6, we provide a constructive methodology for deciding the executability of $\langle \mathfrak{s} \rangle \mathfrak{s}_l$ and for synthesizing a program that can be executed in order to proceed with the analysis and obtain a sound result. The other rules model standard transitions.

Given a program $\mathbf{P} \in \text{Dlmp}$ and a set of initial stores I , we denote by $\mathcal{I} \triangleq \{\mathbf{c} \mid \mathbf{c} = \langle \mathbf{P}, \mathfrak{s} \rangle, \mathfrak{s} \in I\}$ the set of initial states. In sake of simplicity, we consider a partial collecting trace semantics observing only the finite prefixes of finite and infinite execution traces:

$$\mathcal{F}(\mathbf{P}, \mathcal{I}) = \left\{ \mathbf{c}_0 \mathbf{c}_1 \dots \mathbf{c}_n \mid \mathbf{c}_0 \in \mathcal{I}, \forall i < n. \mathbf{c}_i \sim \mathbf{c}_{i+1} \right\}$$

It is known that $\mathcal{F}(\mathbf{P}, \mathcal{I})$ expresses precisely invariant properties of program executions and it can be obtained by fix-point of the following trace set transformer $\mathbf{F} : \wp(\mathbb{C}^*) \longrightarrow \wp(\mathbb{C}^*)$, starting from the set \mathcal{I} of initial configurations, such that

$$\mathcal{F}(\mathbf{P}, \mathcal{I}) = \text{lfp}(\mathbf{F}_{\mathbf{P}, \mathcal{I}}).$$

$$\mathbf{F}_{\mathbf{P}, \mathcal{I}} \triangleq \lambda X. \mathcal{I} \cup \left\{ \mathbf{c}_0 \mathbf{c}_1 \dots \mathbf{c}_i \mathbf{c}_{i+1} \mid \begin{array}{l} \mathbf{c}_0 \mathbf{c}_1 \dots \mathbf{c}_i \in X \\ \mathbf{c}_i \rightsquigarrow \mathbf{c}_{i+1} \end{array} \right\}$$

Finally, we can define the store projection of the partial collecting trace semantics (in the following simply called trace semantics) of a program \mathbf{P} from an initial store $\mathbf{s} \in I$ as

$$\langle \mathbf{P} \rangle \mathbf{s} \triangleq \left\{ \mathbf{s} \mathbf{s}^1 \dots \mathbf{s}^n \mid \begin{array}{l} \exists \mathbf{c}_0 \mathbf{c}_1 \dots \mathbf{c}_n \in \mathcal{F}(\mathbf{P}, \{\mathbf{s}\}). \mathbf{c}_0 = \langle \mathbf{P}, \mathbf{s} \rangle \\ \wedge \forall i \in [1, n]. \exists \mathbf{P}_i \in \mathbf{Dlmp}. \mathbf{c}_i = \langle \mathbf{P}_i, \mathbf{s}^i \rangle \end{array} \right\}$$

Example 4.2. Consider the following *Dlmp* program \mathbf{P} implementing an iterative count by dynamic code modification.

$^1. x := 1; ^2. \text{str} := '\$';$
 $^3. \text{while } x < 3 \{ ^4. \text{str} := 'x := x + 1; ' . \text{str}; ^5. \text{reflect}(\text{str}); \}; ^6. \$$

At each step of computation, let us denote by \mathbf{P} the continuation of the program. A portion of the iterative computation of the collecting semantics, starting from the store \mathbf{s}^0 such that, for each $l \in [1, 6]$, $\mathbf{s}_l^0 = \mathbf{m}_\emptyset$, is reported in Fig. 3. Note that, $\mathbf{s}_1 = \mathbf{m}_\emptyset$ at each step of computation, while $\mathbf{s}_2 = [x/\{1\}, \text{str}/\emptyset]$ after the execution of the first statement. Moreover, in sake of brevity, we will denote as $'s'$ the string $'x := x + 1'$. With a different color, we highlight the execution of *reflect* activating

P	\mathbf{s}_3	\mathbf{s}_4	\mathbf{s}_5	\mathbf{s}_6
$^1. x := 1; ^2. \mathbf{P}$	\mathbf{m}_\emptyset	\mathbf{m}_\emptyset	\mathbf{m}_\emptyset	\mathbf{m}_\emptyset
$^2. \text{str} := '\$'; ^3. \mathbf{P}$	\mathbf{m}_\emptyset	\mathbf{m}_\emptyset	\mathbf{m}_\emptyset	\mathbf{m}_\emptyset
$^3. \text{while } x < 3 \{ ^4. \text{c}; ^6. \$$	$[x/\{1\}, \text{str}/\{' '\}]$	\mathbf{m}_\emptyset	\mathbf{m}_\emptyset	\mathbf{m}_\emptyset
$^4. \text{str} := 'x := x + 1; ' . \text{str}; ^5. \text{c}_1; ^3. \mathbf{P}$	$[x/\{1\}, \text{str}/\{' '\}]$	$[x/\{1\}, \text{str}/\{' '\}]$	\mathbf{m}_\emptyset	\mathbf{m}_\emptyset
$^5. \text{reflect}(\text{str}); ^3. \mathbf{P}$	$[x/\{1\}, \text{str}/\{' '\}]$	$[x/\{1\}, \text{str}/\{' '\}]$	$[x/\{1\}, \text{str}/\{' ', 's'\}]$	\mathbf{m}_\emptyset
$^3. \text{while } x < 3 \{ ^4. \text{c}; ^6. \$$	$[x/\{1, 2\}, \text{str}/\{' ', 's'\}]$	$[x/\{1\}, \text{str}/\{' '\}]$	$[x/\{1\}, \text{str}/\{' ', 's'\}]$	\mathbf{m}_\emptyset
$^4. \text{str} := 'x := x + 1; ' . \text{str}; ^5. \text{c}_1; ^3. \mathbf{P}$	$[x/\{1, 2\}, \text{str}/\{' ', 's'\}]$	$[x/\{1, 2\}, \text{str}/\{' ', 's'\}]$	$[x/\{1\}, \text{str}/\{' ', 's'\}]$	\mathbf{m}_\emptyset
$^5. \text{reflect}(\text{str}); ^3. \mathbf{P}$	$[x/\{1, 2\}, \text{str}/\{' ', 's'\}]$	$[x/\{1, 2\}, \text{str}/\{' ', 's'\}]$	$[x/\{1, 2\}, \text{str}/\{' ', 's', 's'\}]$	\mathbf{m}_\emptyset
$^3. \text{while } x < 3 \{ ^4. \text{c}; ^6. \$$	$[x/\{1, 2, 3, 4\}, \text{str}/\{' ', 's', 's'\}]$	$[x/\{1, 2\}, \text{str}/\{' ', 's'\}]$	$[x/\{1, 2\}, \text{str}/\{' ', 's', 's'\}]$	\mathbf{m}_\emptyset
$^6. \$$	$[x/\{1, 2, 3, 4\}, \text{str}/\{' ', 's', 's'\}]$	$[x/\{1, 2\}, \text{str}/\{' ', 's'\}]$	$[x/\{1, 2\}, \text{str}/\{' ', 's', 's'\}]$	$[x/\{3, 4\}, \text{str}/\{' ', 's', 's'\}]$

Fig. 3. Iterative computation of the collecting semantics of program \mathbf{P} in Example 4.2, with $\mathbf{s} \triangleq x := x + 1$

a new analysis computation, and the memory \mathbf{s}_3 computed by the statements executed by the *reflect*. In particular, the first execution of *reflect*(*str*) is such that $\langle \text{str} \rangle \mathbf{s}_5 \cap \mathbf{Dlmp} = \{x := x + 1; \$\}$. Moreover, the initial store \mathbf{s}^l for the execution of *reflect* is such that $\mathbf{s}_1^l = \mathbf{s}_5$, and $\forall 1 < l \leq l_e \mathbf{s}_l = \mathbf{m}_\emptyset$. $\text{lab}(x := x + 1; \$) = ^1. x := x + 1; ^2. \$$, with $l_e = 2$. Now, the computation of $\text{lab}(x := x + 1; \$)$ is given in Fig. 4 on the left. In this case $\mathbf{s}_{l_e}^e = [x/\{2\}, \text{str}/\{' ', 's'\}]$, hence the new \mathbf{s}_3 is the least upper bound between this $\mathbf{s}_{l_e}^e$ and the previous \mathbf{s}_3 , which is $[x/\{1, 2\}, \text{str}/\{' ', 's'\}]$. The second time *reflect* is executed, we have $\langle \text{str} \rangle \mathbf{s}_5 \cap \mathbf{Dlmp} = \{x := x + 1; \$, x := x + 1; x := x + 1; \$\}$. The calling memory is $\mathbf{s}_1^l = \mathbf{s}_5 = [x/\{1, 2\}, \text{str}/\{' ', 's', 's'\}]$. Similarly to the previous case, the execution of $^1. x := x + 1; ^2. \$$ returns the least upper bound between \mathbf{s}_3 and $[x/\{2, 3\}, \text{str}/\{' ', 's', 's'\}]$ which is $[x/\{1, 2, 3\}, \text{str}/\{' ', 's', 's'\}]$. Finally, the execution of $^1. x := x + 1; ^2. x := x + 1; ^3. \$$ is given in Fig. 4 (on the right). In this case, the resulting memory is the least upper bound between \mathbf{s}_3 and $[x/\{3, 4\}, \text{str}/\{' ', 's', 's'\}]$, which is $[x/\{1, 2, 3, 4\}, \text{str}/\{' ', 's', 's'\}]$, which is also the least upper bound of all the resulting memories, i.e., the new \mathbf{s}_3 .

reflect('s')	s ₁	s ₂	reflect('s; s')	s ₂	s ₃
¹ .x := x + 1; ² .§	[x/{1}, str/{' ', 's'}]	m _∅	¹ .x := x + 1; ² .p	m _∅	m _∅
² .§	[x/{1}, str/{' ', 's'}]	[x/{2}, str/{' ', 's'}]	² .x := x + 1; ³ .§	[x/{2, 3}, str/{' ', 's', 's'}]	m _∅
			³ .§	[x/{2, 3}, str/{' ', 's', 's'}]	[x/{3, 4}, str/{' ', 's', 's'}]

Fig. 4. Some computations of the reflect executions in Example 4.2, with $s \triangleq x := x + 1$.

5 Abstract Interpretation of Strings

5.1 The abstract domain

Let $\mathbb{C}^\sharp = \text{DImp} \times \mathbb{S}^\sharp$ be the domain of abstract states, where $\mathbb{S}^\sharp : \text{PLines}_p \rightarrow \mathbb{M}^\sharp$ denotes abstract stores ranged over by \mathbb{s}^\sharp , and $\mathbb{M}^\sharp : \text{Var} \rightarrow \text{AbstVal}$ denotes the set of abstract memory maps ranged over by \mathbb{m}^\sharp . The domain of abstract values for expressions is $\text{AbstVal} \triangleq \{\top, \text{Interval}, \text{Bool}, \text{FA}_{/\equiv}, \perp\}$ ⁸. It is composed by **Interval**, the standard GC-based abstract domain encoding the interval abstraction of $\wp(\mathbb{Z})$, by **Bool**, the powerset domain of Boolean values, and by $\text{FA}_{/\equiv}$ denoting the domain of FAs up to language equivalence. Given two FA A_1 and A_2 we have that $A_1 \equiv A_2$ iff $\mathcal{L}(A_1) = \mathcal{L}(A_2)$. Hence, the elements of the domain $\text{FA}_{/\equiv}$ are the equivalence classes of FAs recognizing the same language ordered wrt language inclusion $\text{FA}_{/\equiv} = \langle [A]_{\equiv}, \leq_{\text{FA}} \rangle$, where $[A_1]_{\equiv} \leq_{\text{FA}} [A_2]_{\equiv}$ iff $\mathcal{L}(A_1) \subseteq \mathcal{L}(A_2)$. Here concretization is the language recognized \mathcal{L} . By the Myhill-Nerode theorem [19] the domain is well defined and we can use the minimal automata to represent each equivalence class, moreover, the ordering relation is well defined since it does not depend on the choice of the FA used to represent the equivalence class. In particular, we consider the domain $\text{FA}_{/\equiv}$ defined over the finite alphabet Σ , thus, given $A \in \text{FA}_{/\equiv}$, we have that $\mathcal{L}(A) \in \wp(\Sigma^*)$. FAs are closed for finite language intersection and union. They do not form a Galois connection with $\wp(\Sigma^*)$. The finite lub \sqcup_{AbstVal} and glb \sqcap_{AbstVal} among elements of AbstVal are defined as expected: the lub of two abstract values of the same type is given by the lub of the corresponding domain, while the lub between abstract values of different types is \top . This means that, for example, the lub of two intervals is the standard lub over intervals, while the lub between an interval and a FA is \top . Analogously, for the glb returning \perp if applied to different types.

Since **Interval** and $\text{FA}_{/\equiv}$ are not ACC, and, in particular, $\text{FA}_{/\equiv}$ is not closed by lubs of infinite chains, **AbstVal** is also not ACC and not closed. Therefore, we need to define a widening operator ∇ on **AbstVal**. The widening operator among elements of different types returns \top , the widening operator of Boolean elements is the standard lub, **Bool** being ACC, the widening operator between elements of the interval domain is the standard widening operator on **Interval** [14]. Finally, the widening operator on $\text{FA}_{/\equiv}$ is defined in terms of the widening operator ∇_R over finite automata introduced in [21].

Let us consider two FA $A_1 = (Q^1, \delta^1, q_0^1, F^1, \Sigma^1)$ and $A_2 = (Q^2, \delta^2, q_0^2, F^2, \Sigma^2)$ such that $\mathcal{L}(A_1) \subseteq \mathcal{L}(A_2)$: the widening between A_1 and A_2 is formalized in terms

⁸ Note that, we do not consider here implicit type conversion statements, namely each variable, during execution can have values of only one type, nevertheless we consider the reduced product of possible abstract values in order to define only one abstract domain.

Fig. 5. SFT modeling string transformations with $\sigma \in \Sigma$.

of a relation $R \subseteq Q^1 \times Q^2$ between the set of states of the two automata. The relation R is used to define an equivalence relation $\equiv_R \subseteq Q^2 \times Q^2$ over the states of A_2 , such that $\equiv_R = R \circ R^{-1}$. The widening between A_1 and A_2 is then given by the quotient automata of A_2 wrt the partition induced by \equiv_R : $A_1 \nabla_R A_2 = A_2 / \equiv_R$. Thus, the widening operator merges the states of A_2 that are in equivalence relation \equiv_R . By changing the relation R , we obtain different widening operators [21]. It has been proved that convergence is guaranteed when the relation $R_n \subseteq Q^1 \times Q^2$, such that $(q_1, q_2) \in R_n$ if q_1 and q_2 recognize the same language of strings of length at most n [21]. Thus, the parameter n tunes the length of the strings determining the equivalence of states and therefore used for merging them in the widening. It is worth noting that, the smaller is n , the more information will be lost by widening automata. In the following, given two FA A_1 and A_2 with no constraints on the languages they recognize, we define the widening operator parametric on n on $\mathbf{FA}_{/\equiv}$ as follows: $A_1 \nabla_n A_2 \triangleq A_1 \nabla_{R_n} (A_1 \sqcup A_2)$.

5.2 Abstract semantics of expressions

In this section, we model string operations, and in particular we observe that they can be expressed as SFTs, namely as symbolic transformers of a language of strings over Σ . The SFTs that correspond to symbol concatenation $\mathbf{s} \cdot \sigma$ and to substring extraction $\mathbf{substr}(\mathbf{s}, \mathbf{a}_1, \mathbf{a}_2)$ are given in Fig. 5, where σ ranges over the alphabet Σ . In particular, for symbol concatenation we have an SFT T_δ^C for each symbol $\delta \in \Sigma$. Each SFT T_δ^C adds the considered symbol δ at the end of any string (note that if non deterministically we follow the ε edge in the middle of a string then we cannot terminate in a final state anymore, meaning that the input string is not recognized and therefore no output is produced). As far as the sub-string operation is concerned, we have an SFT $T_{n,m}^S$ for each pair of non-negative values n and m , which reads $n - 1$ symbols in the input string without producing outputs, then it reads m symbols from the n -th, releasing the symbol also in output, and finally it reads all the remaining symbols without producing outputs. It is clear that, if the string ends before reaching the starting point n , or before reading m symbols, then the string is not accepted and no output is produced. Namely, if \mathbf{s} is the input string, the transformation works correctly only if $n + m \leq \mathbf{len}(\mathbf{s})$.

We can now define the abstract semantics of expressions as $\langle \mathbf{Exp} \rangle^\# = \mathbb{M}^\# \longrightarrow \mathbf{AbstVal}$ as the best correct approximation of the collecting concrete semantics. For instance, in Fig. ?? we specify the abstract semantics for string expressions. When we perform operations between expressions of the wrong type then we return \top , for example if we add an interval to an FA.

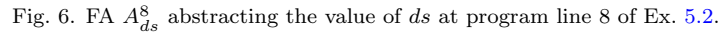
5.3 Abstract Program Semantics

We can now define the abstract transition relation $\leadsto^\# \subseteq \mathbb{C}^\# \times \mathbb{C}^\#$ among abstract states. The rules defining the abstract transition relation can be obtained from the

$$F_{p, \mathcal{I}^\#}^\# \triangleq \lambda X. \mathcal{I}^\# \cup \left\{ c_0^\# \dots c_i^\# c_{i+1}^\# \mid c_0^\# \dots c_i^\# \in X, c_i^\# \rightsquigarrow^\# c_{i+1}^\# \right\}$$

Example 5.2. Consider the following program fragment P

where $ds := \text{deobf}(os)$ is a syntactic sugar for the string transformer in Fig. 6 In Fig. 6-(a) is the FA, namely the abstract value, of ds at program line 8, computed by the proposed static analysis, wrt ∇_3 .



15

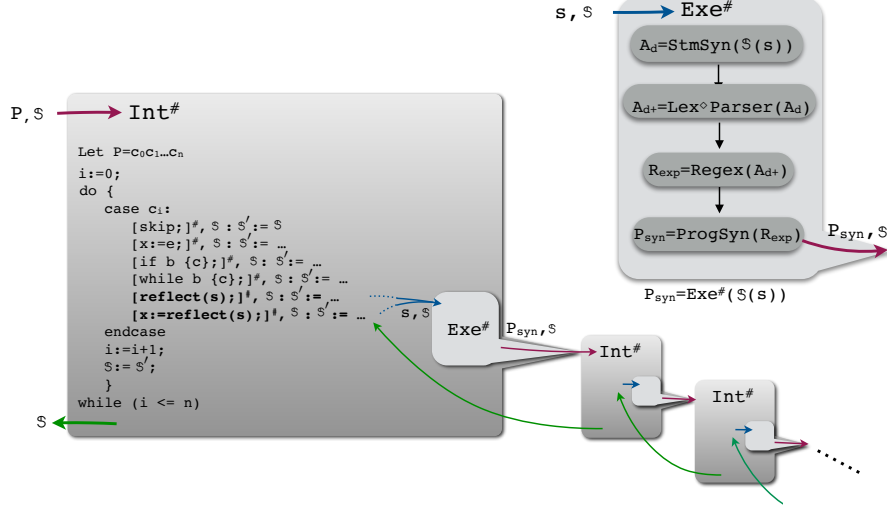


Fig. 7. Architecture and call execution structure of SEA.

means that, our implementation of the analysis needs to approximate the set of executable strings collected during the abstract computation for the arguments of reflection instructions.

6 The SEA Analyzer

The SEA analyser implements both a new string analysis domain and it performs an executability analysis in presence of a reflection statement. SEA is indeed a prototype implementation with the ambition of providing a general language-independent sound-by-construction architecture for the static analysis of self modifying code, where only some components are language-dependent, in our case the abstract interpreter for **DImp**.

The first feature of SEA consists in the implementation of the interpreter based on the flow-sensitive collecting semantics proposed in Sect. 5. The main original contribution is in the way the reflection analysis is handled. In particular, we provide an algorithmic approach for approximating in a decidable way the executability test $\mathcal{L}(\llbracket s \rrbracket^\# s_l^\#) \cap \overline{\text{DImp}}$ and for building a program in **DImp** that soundly approximates the executable programs, i.e., whose semantics soundly approximates the semantics of the code that may be executed in a reflection statement. Our idea is first to filter the automaton collecting the string analysis in order to keep only an over-approximation of the executable strings and then to synthesise a code fragment whose possible executions over-approximate the possible concrete executions. In Fig. 7 we show how SEA works, and we explain the architecture on a running example. In Ex. 5.2 we showed the execution of $\text{Int}^\#(P, s)$, where s starts with *any value* for x , up to program line 8. Now, we can explain how the analysis works. In particular, following the execution structure in Fig. 7, at line 8 we call the execution of $\text{Exe}^\#$ on A_{ds}^8 given in Fig. 6 and in the following simply denoted \bar{A} .

The first step consists in reducing the number of states of the automaton, by

over-approximating every string recognized as a statement, or partial statement, in DImp.

StmSyn.

The idea is to consider the automaton computed by the collecting semantics A , and to collapse all the consecutive edges up to any punctuation symbol in $\{;, \{, \}, \$\}$. In particular, any executable statement will end with $;$, while $\{$ and $\}$ allow to split strings when the body of a **while** or of an **if** begins or ends, finally $\$$ recognises the end of a program. Hence, we design the procedure BUILD, computed by Alg. 2, and recursively called by Alg. 1 that returns an automaton on a finite subset of the alphabet: $\Sigma_{\text{Syn}} = \{;, \$\} \cup \{x; \mid x \in \Sigma^*\} \cup \{x\{ \mid x \in \Sigma^*\}$. In particular, given the parsing tree T_A of the automaton A , obtained by performing a depth first visit on A , we define

$$\text{Str} \triangleq \left\{ x \in (\Sigma \setminus \{;, \{, \}, \$\})^* \mid \begin{array}{l} \exists \text{ path } \pi \text{ in } T_A \text{ such that} \\ x \text{ maximal substring of } \pi \end{array} \right\}.$$

Hence, the finite alphabet of the resulting automaton is $\Sigma_{\text{Syn}}^A = \{;, \$\} \cup \{x; \mid x \in \text{Str}\} \cup \{x\{ \mid x \in \text{Str}\}$.

Algorithm 1 Building the FA.

Require: An FA $A = (Q, \delta, q_0, F, \Sigma)$
Ensure: An FA $A' = (Q', \delta', q_0, F', \Sigma^*)$

- 1: **procedure** StmSyn(A)
- 2: $q'_0 = \delta(q_0, ')$ //The first apex $'$ is erased
- 3: $Q' \leftarrow \{q'_0\}; F' \leftarrow F \cap \{q'_0\}; \delta' \leftarrow \emptyset, \text{Visited} \leftarrow \{q'_0\};$
- 4: STMSYNTR(q'_0);
- 5: **end procedure**
- 6: **procedure** STMSYNTR(q)
- 7: $B \leftarrow \text{BUILD}(A, q);$
- 8: $\text{Visited} \leftarrow \text{Visited} \cup \{q\}; Q' \leftarrow Q' \cup \{p \mid (a, p) \in B\};$
- 9: $F' \leftarrow Q' \cap F; \delta' \leftarrow \delta' \cup \{(q, a, p) \mid (a, p) \in B\};$
- 10: $W \leftarrow \{p \mid (a, p) \in B\} \setminus \text{Visited};$
- 11: **while** $W \neq \emptyset$ **do**
- 12: select p in W ($W \leftarrow W \setminus \{p\}$);
- 13: STMSYNTR(p);
- 14: **end while**
- 15: **end procedure**

The idea of the algorithm is first to reach q'_0 from q_0 reading the symbol $'$, and then to perform, starting from q'_0 , a visit of the states recursively identified by Algorithm 2 and to recursively replace the sequences of edges that recognize a symbol in Σ_{Syn}^A with a single edge labeled by the corresponding string. In particular, from q'_0 we reach the states computed by BUILD(q'_0), and the corresponding read words. Recursively, we apply BUILD to these states, following only those edges that we have not already visited. It is clear that, in this phase all the non-executable strings not ending with a symbol in $\{;, \{, \}, \$\}$ are erased from the automata, hence we have a reduction of non executable strings. For instance, in Fig. 8 we have the computation of StmSyn(\bar{A}), denoted \bar{A}_a . From the computational point of view, we can observe that the procedure BUILD(A, q) executes a number of recursive-call sequences equal to the number of maximal acyclic paths starting from q on A . The

Algorithm 2 Statements recognized from a state q .

Require: An FA $A = (Q, \delta, q_0, F, \Sigma)$
Ensure: I_q set of all pairs (statement, reached state)
1: **procedure** BUILD(A, q)
2: $I_q \leftarrow \emptyset$
3: BUILDTR($q, \varepsilon, \emptyset$)
4: **end procedure**
5: **procedure** BUILDTR($q, \text{word}, \text{Mark}$)
6: $\Delta_q \leftarrow \{(\sigma, p) \mid \delta(q, \sigma) = p\}$
7: **while** $\Delta_q \neq \emptyset$ **do**
8: select (σ, p) in Δ_q ($\Delta_q \leftarrow \Delta_q \setminus \{(\sigma, p)\}$)
9: **if** $(q, p) \notin \text{Mark}$ **then**
10: **if** $\sigma \notin \{;, \{, \}, \$\} \wedge p \notin F$ **then**
11: BUILDTR($p, \text{word}.\sigma, \text{Mark} \cup \{(q, p)\}$)
12: **end if**
13: **if** $\sigma \in \{;, \{, \}, \$\}$ **then** $I_q \leftarrow I_q \cup \{(\text{word}.\sigma, p)\}$
14: **end if**
15: **if** $\sigma = ' \wedge p \in F$ **then** $I_q \leftarrow I_q \cup \{(\text{word}, p)\}$
16: **end if**
17: **end if**
18: **end while**
19: **end procedure**

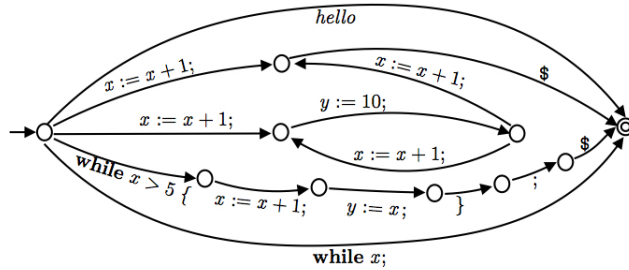


Fig. 8. Automaton $\bar{A}_d = \text{StmSyn}(\bar{A})$.

number of these paths can be computed as $\sum_{q \in Q} (\text{outDegree}(q) - 1) + 1$, where $\text{outDegree}(q)$ is the number of outgoing edges from q . The worst case depth of a recursive-call sequence is $|Q|$. Thus, the worst case complexity of BUILD (when $\text{outDegree}(q) = |Q| \times |\Sigma|$ for all $q \in Q$) is $O(|Q|^3)$. As far as StmSyn is concerned, we can observe that in the worst case we keep in $\text{StmSyn}(A)$ all the $|Q|$ states of A , hence in this case we launch $|Q|$ times the procedure BUILD, and therefore the worst case complexity of StmSyn is $O(|Q|^4)$.

Next step consists in verifying whether the labels of each edge in $\text{StmSyn}(A)$ are potentially executable, or portion of an executable statement.

Lex-Parser.

In order to proceed with the analysis, we need to synthesize a program from A_{d+} approximating the set of executable string values assumed by string s at program line l where reflection is executed. This would allow us to replace the argument of the reflect with the synthesized program and use the same analyser (abstract interpreter) for the analysis of the generated code. Hence, we have to check whether each label in $A_d = \text{StmSyn}(A)$ is in particular in the alphabet $\text{DImp}^- \subseteq \Sigma_{\text{Syn}}^A$ of (partial) statements of DImp statements:

$$\text{Dlmp}^- \triangleq \left\{ \begin{array}{l} \text{skip}; , x := e; , \text{if } b \{ , \text{while } b \{ , \\ \text{reflect}(s); , x := \text{reflect}(s); , \} , \$ \end{array} \right\}$$

where s, e, b are expressions in the language Dlmp . Hence, we need a parser for the language Dlmp^- . This parser can be modelled as the composition of two SFTs. The first one, **Lex**, has to recognise the lexemes in the language by identifying the language tokens. We consider the following set of tokens for Dlmp^- . These tokens correspond to the terminals of Dlmp^- except for the punctuation symbols $\mathfrak{P} \triangleq \{;, \{, \}, \}, \$\}$ that will be directly handled by the parser.

$$\text{Tokens} \triangleq \left\{ \begin{array}{l} \text{id}, \text{const}_s, \text{const}_a, \text{const}_b, \text{aop}, \text{bop}, \text{uop}, \\ \text{num}, \text{len}, \text{conc}_\delta, \text{substr}, \text{relop}, \text{if}, \text{while}, \\ \text{assign}, \text{skip}, \text{reflect}, \text{rand} \end{array} \right\}$$

For each token $T \in \text{Tokens}$, it is possible to define an SFT that recognises its possible lexemes and outputs the lexemes followed by the token name. Let us denote with T_T the SFT that recognises the lexemes of the token $T \in \text{Tokens}$, so, for example, T_{id} is the SFT corresponding to the token id . The transduction is $\mathfrak{T}_{\text{Lex}} : \Sigma^* \longrightarrow (\Sigma \cup \text{Tokens})^*$ defined as:

$$\mathfrak{T}_{\text{Lex}}(\mathbf{a}) \triangleq \left\{ \begin{array}{ll} \mathbf{a}^0 T^0 \mathbf{p}^0 \mathbf{a}^1 T^1 \mathbf{p}^1 \dots \mathbf{a}^n T^n \mathbf{p}^n & \text{if } \mathbf{a} = \mathbf{a}^0 \cdot \mathbf{a}^1 \cdot \dots \cdot \mathbf{a}^n \in \Sigma^* \\ & \forall i \in [0, n] : \mathbf{a}^i \in \mathcal{L}(T_{T_i}), \\ & T_i \in \text{Tokens}, \mathbf{p}^i \in \mathfrak{P}^* \\ \emptyset & \text{otherwise} \end{array} \right.$$

In order to build the **Parser**, we design also the SFT recognising the correct sequences of lexemes and tokens that build respectively arithmetic, boolean and string expressions, and which correctly combines them in order to obtain objects in the language Dlmp^- . Hence, **Parser** should implement the transduction function $\mathfrak{T}_{\text{Parser}} : (\Sigma \cup \text{Tokens})^* \rightarrow \Sigma^*$ is such that: $\mathbf{a} \in \text{Dlmp}^- \Rightarrow \mathfrak{T}_{\text{Parser}}(\mathfrak{T}_{\text{Lex}}(\mathbf{a})) = \mathbf{a}$. This means that the composition $\text{Lex} \diamond \text{Parser}$ allows sequences of Σ_{Syn}^A which are in Dlmp^- . The other implication does not hold since **Parser** allows also sequences of commands of Dlmp^- that contain syntactic errors due an erroneous number of punctuation symbols in \mathfrak{P} . This means that for example the sequence $\text{'xid := assignxid + aoplconsta;; skipskip'}$ is allowed by **Parser** and given in output as it is. In Fig. 9 we can find the automaton \overline{A}_{d+} , which is \overline{A}_{d+} where all the sequences which are not in Dlmp^- are erased.

This module is implemented in SEA using JavaCC [1]: given in input a BNF-style definition of a grammar G it returns as output the parser for G .

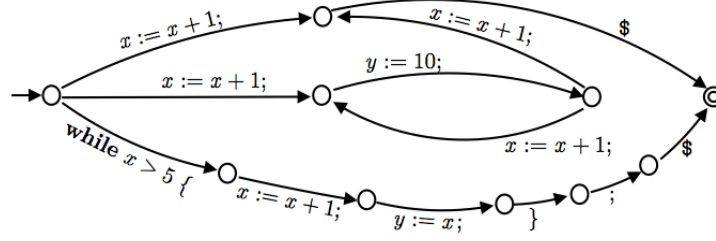


Fig. 9. Executable automaton $\bar{A}_{d+} = \text{Lex} \diamond \text{Parser}(\bar{A}_d)$.

Regex.

The so far obtained automaton can be used to synthesize a program by extracting the regular expression corresponding to the language it recognizes [8]. Let RE be the domain of regular expressions over Dlmp^- , and $\text{Regex} : \text{FA} \rightarrow RE$ be such an extractor. For instance, in the running example, $\bar{R}_{\text{exp}} = \text{Regex}(\bar{A}_{d+})$ is the following regular expression (with standard operators in boldface):

$$\begin{aligned} \bar{R}_{\text{exp}} = & x := x + 1; \$ \text{ } \text{+} \text{ } \text{while } x > 5 \{ x := x + 1; y := x; \}; \$ \\ & + x := x + 1; y := 10; (x := x + 1; y := 10;)^* x := x + 1; \$ \end{aligned}$$

SEA implements the Brzozowski algebraic method [8] to convert an automaton to an equivalent regular expression.

ProgSyn.

Finally, we define **ProgSyn** implementing the function $\wr \cdot \wr_P : RE \rightarrow \text{Dlmp}$ that, given a regular expression $r \in RE$, translates it into a program in **Dlmp**. This is defined in terms of a translation function $\wr \cdot \wr : RE \rightarrow \text{Comm}$ (erasing $\$$) inductively defined on the structure of the regular expression r : Let us denote by d , the symbol d without the last $;$ (e.g., $(x := x + 1); = x := x + 1$)

$$\begin{aligned} \wr d \wr &= d, \text{ if } d \in \text{Dlmp}^- \\ \wr r \$ \wr &= \wr r \wr; \\ \wr r_1 r_2 \wr &= \wr r_1 \wr; \wr r_2 \wr; \\ \wr (r)^* \wr &= \begin{cases} g := \text{rand}(); \\ \text{while } g = 1 \{ \wr r \wr; g := \text{rand}(); \}; \end{cases} \\ \wr r_1 \text{ } \text{+} \text{ } r_2 \wr &= \begin{cases} g := \text{rand}(); \\ \text{if } g = 1 \{ \wr r_1 \wr; \}; \text{ if } g = 2 \{ \wr r_2 \wr; \}; \end{cases} \end{aligned}$$

and $\wr r \wr_P = \text{lab}(\wr r \wr \$)$. Hence, in our running example, the synthesis from the regular expression \bar{R}_{exp} , i.e., $\bar{P}_{\text{syn}} = \text{ProgSyn}(\bar{R}_{\text{exp}})$, is the program

```

1.  $g1 := \mathbf{rand}();$ 
2. if  $g1 = 1$  {3.  $x := x + 1;$  };
4. if  $g1 = 2$  {
5.  $g2 := \mathbf{rand}();$ 
6. if  $g2 = 1$  {7. while  $x > 5$  {8.  $x := x + 1;$  9.  $y := x;$  } };
10. if  $g2 = 2$  {
11.  $x := x + 1;$  12.  $y := 10;$  13.  $g3 = \mathbf{rand}();$ 
14. while  $g3 = 1$  {14.  $x := x + 1;$  15.  $y := 10;$ 
16.  $g3 = \mathbf{rand}();$  };
17.  $x := x + 1;$  };
18. $

```

Soundness.

Next theorem proves the soundness of the approximate program synthesis. Safety (i.e., prefix closed) properties of dynamically generated code are soundly approximated by the synthesized program output of our analysis.

Theorem 6.1. *Let $P \in \mathbf{DImp}$ containing $^l.\mathbf{reflect}(s)$, and let $s \in \mathbb{S}$ be the store on which P is executed. Then, for any $s' \in \mathbb{S}$ such that $s'_1 = s_l$ the partial semantics of any statement in the evaluation of s executed from s' is contained the partial semantics of the synthesized program, formally*

$$\forall c \in \langle s \rangle s_l \cap \overline{\mathbf{DImp}}. \\ \langle c \rangle s' \subseteq \langle \mathbf{ProgSyn}(\mathbf{Regex}(\mathbf{Lex} \diamond \mathbf{Parser}(\mathbf{StmSyn}(A_s^l))) \rangle s'.$$

Termination.

As observed in Example 4.1, the use of reflection suffers from the potential divergence of unbounded nested reflection which goes beyond the control of widening. In this case, the divergence comes directly from the meaning of reflection and cannot be controlled by the semantics once we execute the reflect statement, hence also our analysis in this situation would diverge. SEA ensures soundness until a maximal degree of nested calls to **reflect**.

In order to keep soundness beyond a maximal degree of nested reflections, we can introduce a widening with threshold, i.e., the widening acts after a given number of calls to the abstract interpreter. This corresponds to fix a maximal allowed height of towers, fixing the degree of precision in observing the nesting of reflect statements. Given a *tower height threshold* $\tilde{\tau}$ such that, any tower higher than $\tilde{\tau}$ is approximated as computing any possible value for the program variables whose name is a substring of the string evaluated at $\tilde{\tau}$, therefore guaranteeing soundness. In order to check the height of towers, we need to enrich the store by including a new numerical variable τ counting the nesting level of reflection. Let $S^{\tilde{\tau}} : \mathbf{PLines} \rightarrow \mathbb{M}^{\tilde{\tau}}$ be this enriched domain, where $\mathbb{M}^{\tilde{\tau}} : \mathbf{Var} \cup \{\tau\} \rightarrow \wp(\mathbb{Z}) \cup \mathbf{Bool} \cup \wp(\Sigma^*) \cup \mathbb{Z}$. Hence, we can define a new partial trace semantics $\langle \cdot \rangle^{\tilde{\tau}}$ on the transition system $\langle \mathbb{C}^{\tilde{\tau}}, \sim^{\tilde{\tau}} \rangle$ associated with programs in \mathbf{DImp} , where $\mathbb{C}^{\tilde{\tau}} = \mathbf{DImp} \times S^{\tilde{\tau}}$ is the set of states in the transition system and $\sim^{\tilde{\tau}} \subseteq \mathbb{C}^{\tilde{\tau}} \times \wp(\mathbb{C}^{\tilde{\tau}})$ is the transition relation. Note that, the semantics of all statements does not change (supposing that $m_{\emptyset} \in \mathbb{M}^{\tilde{\tau}}$ associates 0 to τ) except for **reflect**, whose new rule should count the number of recursive interpreter $\mathbf{Int}^{\#}$ calls.

P	TAJS analysis of y	TAJS reflection of y
$y := 'x = x + 1;'; \mathbf{reflect}(y);$	$'x = x + 1;'$	$x := x + 1;$
$\mathbf{if } x > 0 \{ y := 'a := a + 1;';$ $\mathbf{if } x < 0 \{ y := 'b := b + 1;'; \mathbf{reflect}(y);$	String	AnalysisLimitationException
$x := 1; y := '';$ $\mathbf{while } x < 3 \{ y := y \cdot 'x := x + 1;'; x := x + 1; \};$ $y := y \cdot \$; \mathbf{reflect}(y);$	String	AnalysisLimitationException

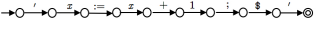
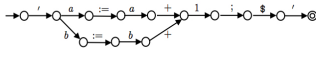
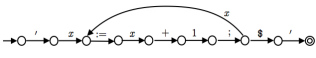
P	SEA analysis of y	SEA reflection of y
$y := 'x = x + 1;'; \mathbf{reflect}(y);$		$x := x + 1; \$$
$\mathbf{if } x > 0 \{ y := 'a := a + 1;';$ $\mathbf{if } x < 0 \{ y := 'b := b + 1;'; \mathbf{reflect}(y);$		$g1 := \mathbf{rand}();$ $\mathbf{if } g1 = 1 \{ a := a + 1; \};$ $\mathbf{if } g1 = 2 \{ b := b + 1; \}; \$$
$x := 1; y := '';$ $\mathbf{while } x < 3 \{ y := y \cdot 'x := x + 1;'; x := x + 1; \};$ $y := y \cdot \$; \mathbf{reflect}(y);$		$x := x + 1; g1 := \mathbf{rand}();$ $\mathbf{while } g1 = 1 \{$ $\quad x := x + 1; g1 := \mathbf{rand}(); \}; \$$

Fig. 10. SEA vs TAJs

7 Evaluation

SEA is a proof of concept, showing that it is possible to design and implement an efficient sound-by-construction static analyser based on abstract interpretation for self modifying code written in high-level script-like languages. It was not in the intention of SEA to be optimal and directly applicable to existing script dynamic languages, such as PHP or JavaScript. We implemented SEA in Java 1.8 and we tested it on some significant code examples in order to highlight the strengths and the weaknesses of the analyser presented. In particular, we evaluate the precision of our string abstract domain as compared to TAJs [27,31,2] (version 0.9-8), which is one of the best static analyser available for JavaScript based on abstract interpretation. To the best of our knowledge TAJs is the only tool statically analysing string-to-code primitives such as **eval**. This approach basically consist of a sound transformation of a JavaScript program $P_{\mathbf{eval}}$, containing **eval**, in another JavaScript program $P_{\mathbf{uneval}}$ where the **eval** statement is substituted with its argument, obviously converted in executable code, when this is possible, namely when the code to execute can be statically extracted as a constant form $P_{\mathbf{eval}}$. All examples in the next sections have been compiled from DImp into a semantics-equivalent JavaScript program, in order to perform the comparison with TAJs.

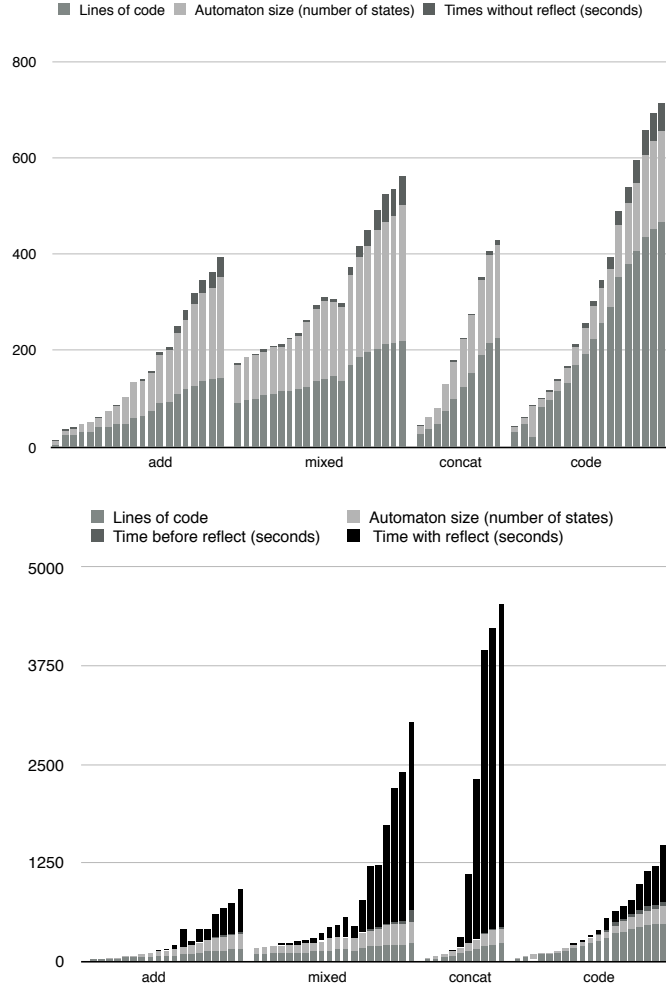


Fig. 11. Execution times in secs. without reflection (top) and with reflection (bottom). We ran the tests on an Intel i5-4210u 2.20 GHz processor.

7.1 Precision

We performed more than 100 tests on programs of variable length, 70 of them are used to test the SEA performances and will be addressed in Sect. 7.2. We observed that the results can be classified in three different classes depending on some features of the analysed program. We report three significant examples in Fig. 10 where we also compare SEA with TAJIS.

The first class of tests consists in all the programs where the string variables collect only one value during execution, i.e., they are constant string variables. A toy example in this class of programs is provided in the first row of Fig. 10, where the string value contained in y is hard-coded and constant. In this case, both SEA and TAJIS are precise and no loss of information occurs during the analyses. By using the value of y in SEA as input of **reflect**, we obtain exactly the statement $x := x + 1$; since Exe^\sharp behaves as the identity function. TAJIS performs the uneval transformation and executes the same statement.

The second class of tests consists in all the programs where there are no constant string variables, namely variables whose value before the reflection is not precisely known being a set of potential string values. As toy example of this class of programs, consider the snippet of code in the second row of Fig. 10, a simplification of Example 5.2. In this case, since we don't have any information about x we must consider both the branches, which means that before the reflection we only know that y is one value between ' $a := a + 1$ ' and ' $b := b + 1$ '.

If we analyse this program in TAJ, we observe that, after the if statement, the value of y is identified as a string, since TAJ does not perform a collecting semantics and when it loses the constant information it loses the whole value. Unfortunately, this loss of precision, in the analysis of y , makes the TAJ analysis stuck, producing an exception when the reflection statement is called on the non constant variable. On the other hand, SEA computes the collecting semantics and therefore it keeps the least upper bound of the stores computed in each branch, obtaining the abstract value for y modelled by the automaton A_y in the second row equivalent to the regular expression ' $a := a + 1; ' + 'b := b + 1; '$ '. Afterwards, the SEA analyser returns and analyse the sound approximation of the program passed to the reflection statement reported in the second row, which is the result of $\text{Exe}^\#(A_y)$.

In the last class of examples, the string that will be executed is not constant and it is dynamically built during execution. In the simple example provided in Fig. 10 the dynamically generated statement is ' $x := x + 1; x := x + 1; '$ '. In this case, as it happened before, TAJ loses the value of y (which is a set of potential strings) and can only identify y as a string. This means that, again, the reflection statement makes the analysis stuck, throwing an exception. On the other hand, SEA performs a sound over-approximation of the set of values computed in y . In particular, the analysis, in order to guarantee termination and therefore decidability, computes widening instead of least upper bound between automata inside the loop. This clearly introduces imprecision, since it makes us lose the control on the number of iterations. In particular, instead of computing the precise automaton containing only and all the possible string values for y (as in the previous case) we compute an automaton strictly containing the concrete set of possible string values. The computed automaton is reported in the third row and it is equivalent to the regular expression ' $x := x + 1; (x := x + 1)^* '$ '. The presence of possible infinite sequences of ' $x := x + 1; '$ ' is due to the over-approximation induced by the widening operator ∇_3 on automata. Nevertheless, note that the widening parameter can be chosen by the user in order to tune the desired precision degree of the analysis: the higher is the parameter the more precise and costly is the analysis. It should be clear that, the introduced loss of precision increases the imprecision in any further analyses which uses the synthesised code. The synthesis of the program from the abstract value of y returns the code reported in the third row: due to widening, as observed above, the command ' $x := x + 1$ ' can diverges.

A final observation on precision concerns the analysis of programs with unknown inputs. SEA considers an unknown input as a variable that may assume any possible string value. It is clear that in this kind of situations, TAJ necessarily get

stuck whenever something depending on this unknown input is executed. Instead, SEA can keep some information since the abstract value consisting in *any possible value* is modelled by the automaton recognising Σ^* . In this way SEA can trace the string manipulations (substring and concatenation) on the unknown input, occurring during the execution.

7.2 Performances

We have tested the performances of SEA on a benchmark of 70 increasingly complex programs. Each program manipulates an automaton and finally it reflects the string value. The benchmark can be clustered in four families depending on the kind of string operations considered in the programs, determining the kind of automata manipulations performed by the analysis: **add** (programs where the manipulation of strings adds always new whole statements, this corresponds, in our analysis, to adding completely new paths in the automaton), **concat** (programs where the manipulation of strings concatenates new paths to those in the automaton), **mixed** (programs performing both the manipulations), **code** (programs in the **add** family, where we have added statements not manipulating strings, i.e., standard code not affecting strings).

Fig. 11-top, shows the results obtained from the benchmark concerning the string analysis without reflection: increasing the lines of code as well as the number of the automaton states, the execution time increases with an almost linear trend.

In Fig 11-bottom, we show the results due to string executability analysis. The total execution time increases more quickly of both than the length code and the automata dimension. But, it is worth noting that most of the time increase is due to the execution of the code generated by $\text{Exe}^\#$ (the top black portion of the bars in Fig. 11-bottom), while the time cost of the analysis still increases with an almost linear trend. This outcome tells us that the string and executability analysis scale quite well on the source original code, but it gets worse on the code generated by $\text{Exe}^\#$. We believe that this is due to the implementation of the synthesised code, which does not optimise the code generated by $\text{Exe}^\#$. The optimisation of the generated code is a future work that deserves further investigation.

7.3 Analysis limitations and conclusions

SEA attacks an extremely hard problem in static program analysis, providing the very first proof of concept in sound static analysis for self-modifying code based on bounded reflection in a high-level script-like language. The main limitations of SEA are two: the simplicity of the programming language analysed, missing some important language features such as procedure calls and objects, and the fact that **Dlmp** is not a real script language. The choice of keeping the language as simple as possible is due to the aim of designing a core analyser focusing mainly on string executability analysis. We are conscious that, in order to implement a real-world static analyser we will have to integrate in our language several more sophisticated language features, but this is beyond this proof of concept. For instance, an extension

would be the possibility of allowing implicit type conversion statements provided by many modern languages, such as PHP, JavaScript or Python.

As far as the second limitation is concerned, we already observed that this choice is due to the ambition of providing the most general possible architecture for executability string analysis. We believe that SEA is a step towards an implementation of a sound-by-construction analyser for reflection in real dynamic languages, since its design is fully language independent. In particular, the SEA architecture is invariant on the choice of the string abstraction, in our case FA, as well as the other state abstractions, in our case intervals, and on the language features, provided that their formal semantics is given.

References

- [1] JavaCC - the Java Compiler Compiler. <https://javacc.java.net/>. Accessed: 2016-11-13.
- [2] TAJs - Type Analyzer for JavaScript. <https://github.com/cs-au-dk/TAJS>. Accessed: 2016-11-13.
- [3] AN, J. D., CHAUDHURI, A., FOSTER, J. S., AND HICKS, M. Dynamic inference of static types for Ruby. In *POPL* (2011), T. Ball and M. Sagiv, Eds., ACM, pp. 459–472.
- [4] ANCKAERT, B., MADOU, M., AND BOSSCHERE, K. D. A model for self-modifying code. In *Information Hiding* (2006), J. Camenisch, C. S. Collberg, N. F. Johnson, and P. Sallee, Eds., vol. 4437 of *LNCS*, Springer, pp. 232–248.
- [5] BESSEY, A., BLOCK, K., CHELF, B., CHOU, A., FULTON, B., HALLEM, S., GROS, C., KAMSKY, A., MCPEAK, S., AND ENGLER, D. R. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75.
- [6] BIGGAR, P., AND GREGG, D. Static analysis of dynamic scripting languages. Technical report, Department of Computer Science, Trinity College Dublin, 2009.
- [7] BODDEN, E., SEWE, A., SINSCHKE, J., OUESLATI, H., AND MEZINI, M. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proc. of the 33rd Internat. Conf. on Software Engineering, ICSE 2011* (2011), pp. 241–250.
- [8] BRZOWSKI, J. A. Derivatives of regular expressions. *J. ACM* 11, 4 (1964), 481–494.
- [9] CAI, H., SHAO, Z., AND VAYNBERG, A. Certified self-modifying code. In *PLDI* (2007), J. Ferrante and K. S. McKinley, Eds., ACM, pp. 66–77.
- [10] CHRISTENSEN, A. S., MØLLER, A., AND SCHWARTZBACH, M. I. Precise analysis of string expressions. In *Static Analysis, 10th International Symposium, SAS 2003, San Diego, CA, USA, June 11-13, 2003, Proceedings* (2003), R. Cousot, Ed., vol. 2694 of *Lecture Notes in Computer Science*, Springer, pp. 1–18.
- [11] CHUGH, R., MEISTER, J. A., JHALA, R., AND LERNER, S. Staged information flow for JavaScript. In *PLDI* (2009), M. Hind and A. Diwan, Eds., ACM, pp. 50–62.
- [12] COUSOT, P. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.* 277, 1-2 (2002), 47–103.
- [13] COUSOT, P., AND COUSOT, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages (POPL '77)* (1977), ACM Press, pp. 238–252.
- [14] COUSOT, P., AND COUSOT, R. Systematic design of program analysis frameworks. In *Conference Record of the 6th ACM Symposium on Principles of Programming Languages (POPL '79)* (1979), ACM Press, pp. 269–282.
- [15] COUSOT, P., AND COUSOT, R. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the Seventh ACM Conference on Functional Programming Languages and Computer Architecture* (25–28 June 1995), ACM Press, New York, NY, pp. 170–181.
- [16] DALLA PREDÀ, M., GIACOBazzi, R., AND DEBRAY, S. K. Unveiling metamorphism by abstract interpretation of code properties. *Theor. Comput. Sci.* 577 (2015), 74–97.

- [17] DALLA PREDÀ, M., GIACOBBAZI, R., AND MASTROENI, I. Completeness in approximated transductions. In *Static Analysis, 23rd International Symposium, SAS 2016*. (2016), vol. 9837 of *LNCS*, pp. 126–146.
- [18] DANVY, O., AND MALMKJÆR, K. Intensions and extensions in a reflective tower. In *LISP and Functional Programming* (1988), pp. 327–341.
- [19] DAVIS, M., SIGAL, R., AND WEYUKER, E. J. *Computability, Complexity, and Languages, Second Edition: Fundamentals of Theoretical Computer Science (Computer Science and Scientific Computing) 2nd edition*. Elsevier, 1994.
- [20] DOH, K., KIM, H., AND SCHMIDT, D. A. Abstract parsing: Static analysis of dynamically generated string output using lr-parsing technology. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings* (2009), J. Palsberg and Z. Su, Eds., vol. 5673 of *Lecture Notes in Computer Science*, Springer, pp. 256–272.
- [21] D’SILVA, V. Widening for automata. Diploma Thesis, Institut Fur Informatik, Universitat Zurich, 2006.
- [22] GIACOBBAZI, R. Abductive analysis of modular logic programs. *J. of Logic and Computation* 8, 4 (1998), 457–484.
- [23] HEINTZE, N., AND JAFFAR, J. Set constraints and set-based analysis. In *Principles and Practice of Constraint Programming, Second International Workshop, PPCP’94, Rosario, Orcas Island, Washington, USA, May 2-4, 1994, Proceedings* (1994), A. Borning, Ed., vol. 874 of *Lecture Notes in Computer Science*, Springer, pp. 281–298.
- [24] HOOIMEIJER, P., LIVSHITS, B., MOLNAR, D., SAXENA, P., AND VEANES, M. Bek: Modeling imperative string operations with symbolic transducers. Tech. Rep. MSR-TR-2010-154, November 2010.
- [25] HOOIMEIJER, P., LIVSHITS, B., MOLNAR, D., SAXENA, P., AND VEANES, M. Fast and precise sanitizer analysis with BEK. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings* (2011), USENIX Association.
- [26] JENSEN, S. H., JONSSON, P. A., AND MØLLER, A. Remedying the eval that men do. In *International Symposium on Software Testing and Analysis, ISSSTA 2012, Minneapolis, MN, USA, July 15-20, 2012* (2012), M. P. E. Heimdahl and Z. Su, Eds., ACM, pp. 34–44.
- [27] JENSEN, S. H., MØLLER, A., AND THIEMANN, P. Type Analysis for JavaScript. In *Static Analysis, 16th International Symposium, SAS 2009, Los Angeles, CA, USA, August 9-11, 2009. Proceedings* (2009), pp. 238–255.
- [28] KIM, H., DOH, K., AND SCHMIDT, D. A. Static validation of dynamically generated HTML documents based on abstract parsing and semantic processing. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings* (2013), F. Logozzo and M. Fähndrich, Eds., vol. 7935 of *Lecture Notes in Computer Science*, Springer, pp. 194–214.
- [29] MAVROGIANNOPOULOS, N., KISSERLI, N., AND PRENEEL, B. A taxonomy of self-modifying code for obfuscation. *Computers & Security* 30, 8 (2011), 679–691.
- [30] MINAMIDE, Y. Static approximation of dynamically generated web pages. In *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005* (2005), A. Ellis and T. Hagino, Eds., ACM, pp. 432–441.
- [31] MØLLER, A. Static Analysis of JavaScript. In *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Invited talk* (2015).
- [32] RICHARDS, G., HAMMER, C., BURG, B., AND VITEK, J. The eval that men do - A large-scale study of the use of eval in javascript applications. In *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings* (2011), M. Mezini, Ed., vol. 6813 of *Lecture Notes in Computer Science*, Springer, pp. 52–78.
- [33] SMITH, B. C. Reflection and semantics in lisp. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah, USA, January 1984* (1984), K. Kennedy, M. S. V. Deussen, and L. Landweber, Eds., ACM Press, pp. 23–35.
- [34] THIEMANN, P. Grammar-based analysis of string expressions. In *Proceedings of TLDI’05: 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Long Beach, CA, USA, January 10, 2005* (2005), J. G. Morrisett and M. Fähndrich, Eds., ACM, pp. 59–70.
- [35] VEANES, M., HOOIMEIJER, P., LIVSHITS, B., MOLNAR, D., AND BJØRNER, N. Symbolic finite state transducers: algorithms and applications. In *POPL* (2012), J. Field and M. Hicks, Eds., ACM, pp. 137–150.
- [36] VENET, A. Automatic analysis of pointer aliasing for untyped programs. *Sci. Comput. Program.* 35, 2 (1999), 223–248.

- [37] WAND, M., AND FRIEDMAN, D. P. The mystery of the tower revealed: A nonreflective description of the reflective tower. *Lisp and Symbolic Computation* 1, 1 (1988), 11–37.
- [38] WANG, X., JHI, Y., ZHU, S., AND LIU, P. Still: Exploit code detection via static taint and initialization analyses. In *ACSAC* (2008), IEEE Computer Society, pp. 289–298.
- [39] XIE, Y., AND AIKEN, A. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006* (2006), A. D. Keromytis, Ed., USENIX Association.
- [40] YU, F., ALKHALAF, M., AND BULTAN, T. Patching vulnerabilities with sanitization synthesis. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011* (2011), R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds., ACM, pp. 251–260.
- [41] ZHAUNIAROVICH, Y., AHMAD, M., GADYATSKAYA, O., CRISPO, B., AND MASSACCI, F. Stadya: Addressing the problem of dynamic code updates in the security analysis of android applications. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy* (2015), CODASPY '15, ACM, pp. 37–48.